[第4章 Kafka 204](#_Toc515544318)

[4.1 kafka的介绍 204](#_Toc515544319)

[4.2 kafka的架构 207](#_Toc515544320)

[4.3 kafka集群的安装与搭建 210](#_Toc515544321)

[4.4 kafkfa的原理 213](#_Toc515544322)

[4.5 kafka的API使用 216](#_Toc515544323)

[4.6 kafka与其他的整合使用 220](#_Toc515544324)

# Kafka

## kafka的介绍

#### 1什么是kafka？



Apache Kafka是一个开源**消息**系统，由Scala写成。是由Apache软件基金会开发的一个开源消息系统项目。

Kafka最初是由LinkedIn开发，并于2011年初开源。2012年10月从Apache Incubator毕业。该项目的目标是为处理实时数据提供一个统一、高通量、低等待的平台。

**Kafka是一个分布式消息队列：生产者、消费者的功能。它提供了类似于JMS的特性，但是在设计实现上完全不同，此外它并不是JMS规范的实现。**

Kafka对消息保存时根据Topic进行归类，发送消息者称为Producer,消息接受者称为Consumer,此外kafka集群有多个kafka实例组成，每个实例(server)成为broker。

无论是kafka集群，还是producer和consumer都依赖于**zookeeper**集群保存一些meta信息，来保证系统可用性

#### 2 kafka与传统消息系统的区别

###### 1、在[架构](http://lib.csdn.net/base/architecture)模型方面，

RabbitMQ遵循AMQP协议，RabbitMQ的broker由Exchange,Binding,queue组成，其中exchange和binding组成了消息的路由键；客户端Producer通过连接channel和server进行通信，Consumer从queue获取消息进行消费（长连接，queue有消息会推送到consumer端，consumer循环从输入流读取数据）。rabbitMQ以broker为中心；有消息的确认机制。

kafka遵从一般的MQ结构，producer，broker，consumer，以consumer为中心，消息的消费信息保存的客户端consumer上，consumer根据消费的点，从broker上批量pull数据；无消息确认机制。

###### 2、在吞吐量，

kafka具有高的吞吐量，内部采用消息的批量处理，zero-copy机制，数据的存储和获取是本地磁盘顺序批量操作，具有O(1)的复杂度，消息处理的效率很高。

rabbitMQ在吞吐量方面稍逊于kafka，他们的出发点不一样，rabbitMQ支持对消息的可靠的传递，支持事务，不支持批量的操作；基于存储的可靠性的要求存储可以采用内存或者硬盘。

###### 3、在可用性方面，

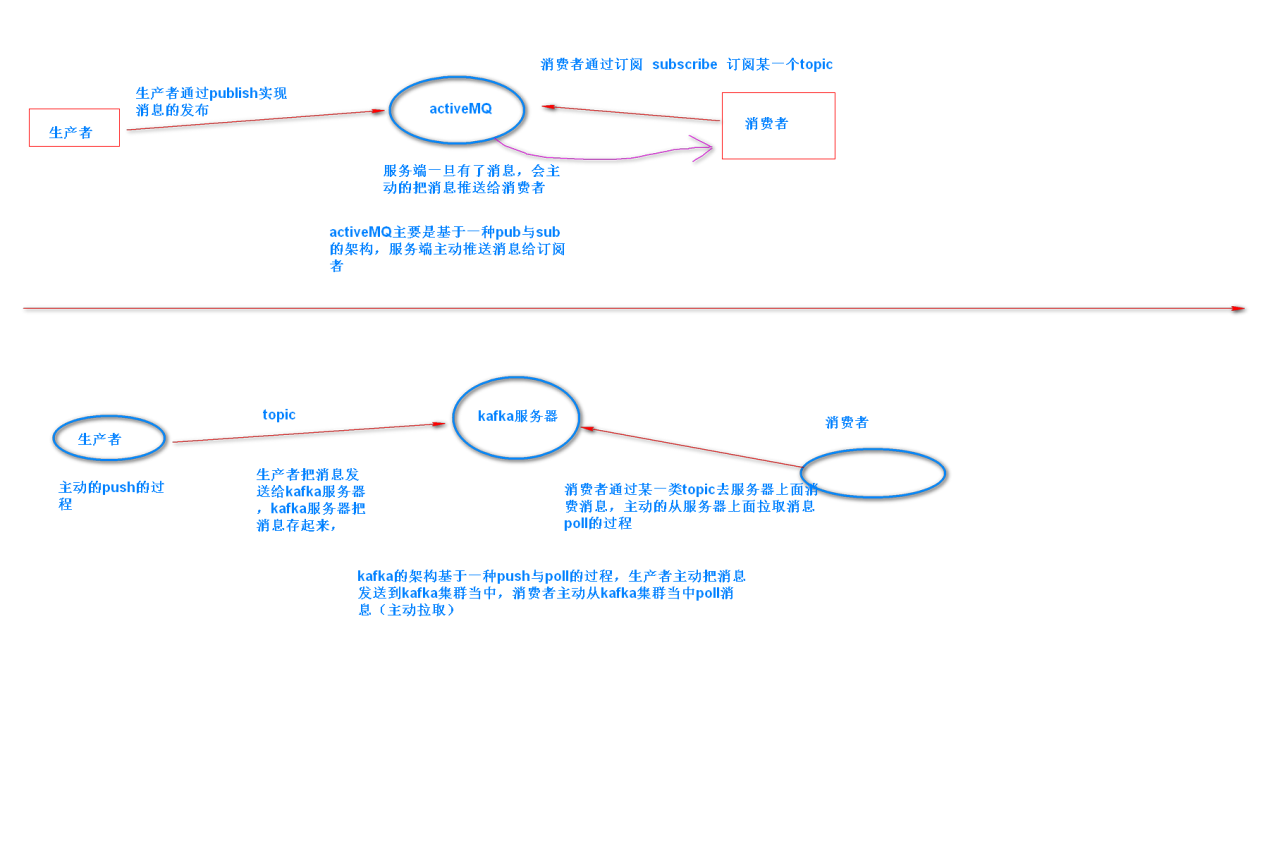
rabbitMQ支持miror的queue，主queue失效，miror queue接管。

kafka的broker支持主备模式。

###### 4、在集群负载均衡方面

kafka采用zookeeper对集群中的broker、consumer进行管理，可以注册topic到zookeeper上；通过zookeeper的协调机制，producer保存对应topic的broker信息，可以随机或者轮询发送到broker上；并且producer可以基于语义指定分片，消息发送到broker的某分片上。

#### 3 kafka与activeMQ的区别



Topic:主题,即一个标识,类似于map里面的key,通过它来给消息分类,消息根据Topic进行归类。

**共同点**:都有生产者和消费者两大组件,生产者发送消息给各自的服务器,(发送消息是就会定义一个topic)并进行存储。

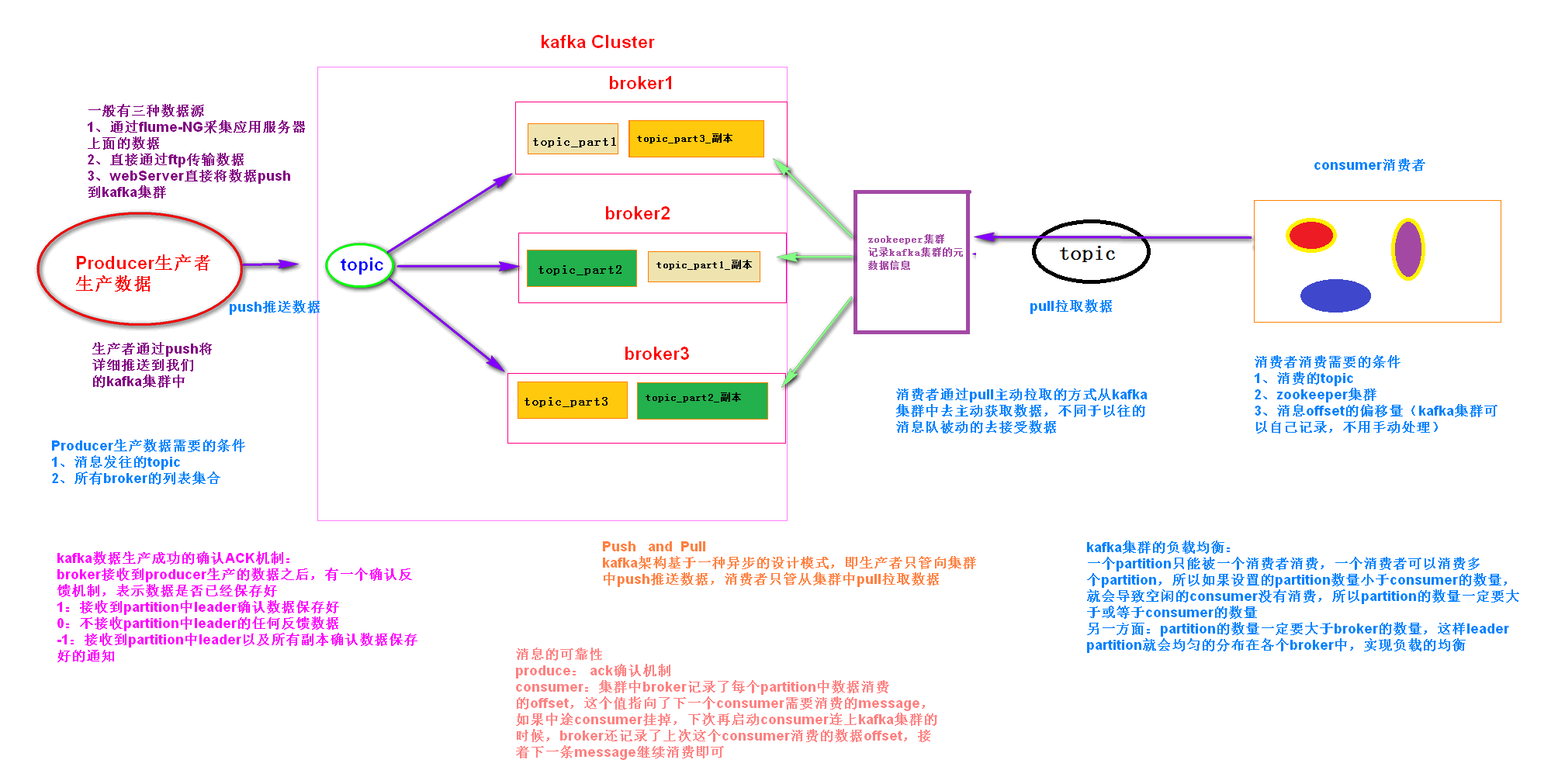
**不同点**:

activeMQ:消费者会提前订阅自己需要的topic，当该topic中有了消息以后,activeMQ服务器会发送消息给消费者,然后消费者再去服务器中拿到自己想要的数据。

Kafka:消费者(指定topic)会定时去kafka服务器中拿该topic中的数据。

## kafka的架构

基于producer consumer topic broker 等的一个基本架构



#### kafka的组件介绍

producer：生产者，主要用于我们的消息的生产，通过producer将我们的消息push到kafka集群当中

topic：某一类消息的高度抽象，可以理解成某一类消息的集合,一类消息，每个topic将被分成多个partition(区)，在集群的配置文件中配置。

broker：kafka的服务器，一个broker就代表一个服务器的节点

partition：分区的概念，一个topic当中的消息，可以拆分成多个partition分区，存放在多个不同的服务器上，实现数据存放的横向扩展

repliaction：副本，所有的partition都可以指定存放几个副本，做到数据的冗余，保证数据的安全

segment：每个partiiton由多个segment组成，segment又包含了两部分，一个.log文件，一个是.index文件

.log：存放我们的日志文件，所有的数据，最后都以日志文件的形式存放到了kafka集群当中

.index ：索引文件，所有的.log文件的索引都存放在了这里，便于我们查找某一条日志文件的快速

consumer：消费者，消费我们kafka集群当中的消息，

问题：如何知道消费者消费到了哪一条消息来了？？？

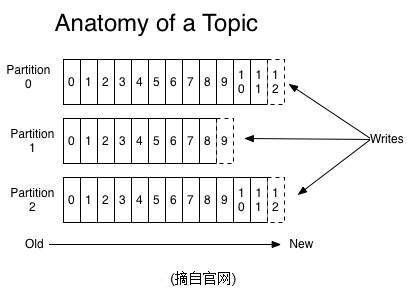
可以通过记录的方式，记下来每次我们消费的位置

第一种记录方式：kafka的本地文件系统，比较慢，对应kafka的一个慢速消费的方式

第二种记录方式：zookeeper当中的节点数据记录，比较快，对应kafka的一个快速消费的方式

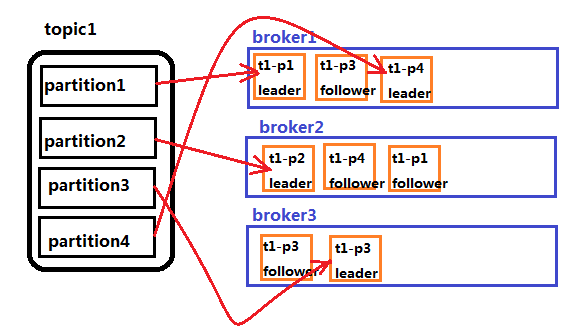
offset：偏移量，就是记录的我们消费到了哪一条数据来了。

发布者发到某个topic的消息会被均匀的分布到多个part上，broker收到发布消息往对应part的最后一个segment上添加该消息。



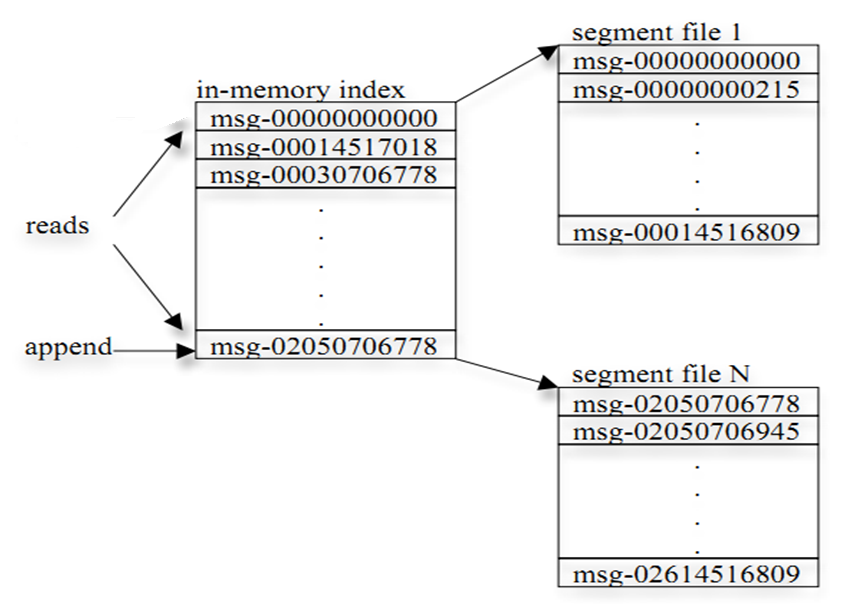
#### partition分布

1. partitions分区到不同的server上，一个partition保存在一个server上，避免一个server上的文件过大，同时可以容纳更多的consumer消费,有效提升并发消费的能力。
2. 这个server(如果保存的是partition的leader)负责partition的读写。可以配置备份。
3. 每个partition都有一个server为"leader"，负责读写，其余的相对备份机为follower，follower同步leader数据，负责leader死了之后的接管。n个leader均衡的分散在每个server上。
4. partition的leader和follower之间监控通过zookeeper完成。



#### segment

1. 每个segment中存储多条消息，消息id由其逻辑位置决定，即从消息id可直接定位到消息的存储位置，避免id到位置的额外映射。
2. 当某个segment上的消息条数达到配置值或消息发布时间超过阈值时，segment上的消息会被flush到磁盘，只有flush到磁盘上的消息订阅者才能订阅到
3. segment达到一定的大小（可以通过配置文件设定,默认1G）后将不会再往该segment写数据，broker会创建新的segment。



#### offset

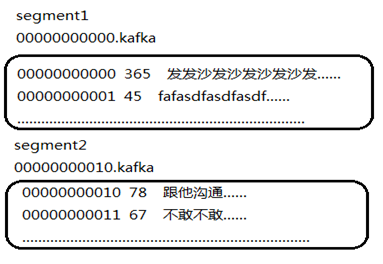
offset是每条消息的偏移量。

segment日志文件中保存了一系列"log entries"(日志条目),每个log entry格式为"4个字节的数字N表示消息的长度" + "N个字节的消息内容";

每个日志文件都有一个offset来唯一的标记一条消息,offset的值为8个字节的数字,表示此消息在此partition中所处的起始位置.

每个partition在物理存储层面,有多个log file组成(称为segment).

segment file的命名为"最小offset".log.例如"00000000000.log";其中"最小offset"表示此segment中起始消息的offset.



## kafka集群的安装与搭建

kafka的官网

<http://kafka.apache.org/downloads>

下载地址：

<https://www.apache.org/dyn/closer.cgi?path=/kafka/1.0.0/kafka_2.11-1.0.0.tgz>

第一步：下载上传压缩包

第二步：解压

tar -zxvf kafka\_2.11-1.0.0.tgz -C ../servers/

第三步：安装包的分发

第四步：修改配置文件

cd /export/servers/kafka\_2.11-1.0.0/config

vim server.properties

df -lh 查看磁盘空间

第一台服务器修改配置文件

broker.id=0

num.network.threads=3

num.io.threads=8

socket.send.buffer.bytes=102400

socket.receive.buffer.bytes=102400

socket.request.max.bytes=104857600

log.dirs=/export/servers/kafka\_2.11-1.0.0/logs

num.partitions=2

num.recovery.threads.per.data.dir=1

offsets.topic.replication.factor=1

transaction.state.log.replication.factor=1

transaction.state.log.min.isr=1

log.flush.interval.messages=10000

log.flush.interval.ms=1000

log.retention.hours=168

log.segment.bytes=1073741824

log.retention.check.interval.ms=300000

zookeeper.connect=node01:2181,node02:2181,node03:2181

zookeeper.connection.timeout.ms=6000

group.initial.rebalance.delay.ms=0

delete.topic.enable=true

host.name=node01

第二台服务器修改配置文件

broker.id=1

num.network.threads=3

num.io.threads=8

socket.send.buffer.bytes=102400

socket.receive.buffer.bytes=102400

socket.request.max.bytes=104857600

log.dirs=/export/servers/kafka\_2.11-1.0.0/logs

num.partitions=2

num.recovery.threads.per.data.dir=1

offsets.topic.replication.factor=1

transaction.state.log.replication.factor=1

transaction.state.log.min.isr=1

log.flush.interval.messages=10000

log.flush.interval.ms=1000

log.retention.hours=168

log.segment.bytes=1073741824

log.retention.check.interval.ms=300000

zookeeper.connect=node01:2181,node02:2181,node03:2181

zookeeper.connection.timeout.ms=6000

group.initial.rebalance.delay.ms=0

delete.topic.enable=true

host.name=node02

第三台服务器修改配置文件

broker.id=2

num.network.threads=3

num.io.threads=8

socket.send.buffer.bytes=102400

socket.receive.buffer.bytes=102400

socket.request.max.bytes=104857600

log.dirs=/export/servers/kafka\_2.11-1.0.0/logs

num.partitions=2

num.recovery.threads.per.data.dir=1

offsets.topic.replication.factor=1

transaction.state.log.replication.factor=1

transaction.state.log.min.isr=1

log.flush.interval.messages=10000

log.flush.interval.ms=1000

log.retention.hours=168

log.segment.bytes=1073741824

log.retention.check.interval.ms=300000

zookeeper.connect=node01:2181,node02:2181,node03:2181

zookeeper.connection.timeout.ms=6000

group.initial.rebalance.delay.ms=0

delete.topic.enable=true

host.name=node03

第四步：三台服务器的启动

三台服务器的启动

nohup bin/kafka-server-start.sh config/server.properties >/dev/null 2>&1 &

## kafkfa的原理

#### 4.4.1数据生产分发策略

第一种：如果指定了分区号，那么数据就会全部进入到指定的分区里面去

//producer.send(new ProducerRecord<String, String>("test",1,"1", "hello world"+i)); //ProducerRecord 使用两个形参，第一个形参是我们的topic主题，第二个参数就是我们需要发送的消息

如果给了分区号，也给了key值，那么优先使用指定的分区号

第二种：如果没有给定分区号，但是给了数据的key，那么通过key的hash取值来决定数据到哪一个分区里面去

producer.send(**new** ProducerRecord<String, String>(**"test"**,**"101"**, **"hello world"**+i)); *//ProducerRecord 使用两个形参，第一个形参是我们的topic主题，第二个参数就是我们需要发送的消息*

如果没有指定分区号，给定了我们的key，那么就会通过key的hash取值进行分区，实际工作当中，如果通过这种方式进行分区一定要注意，key的值一定要变化

第三种：没有给定分区号，也没有给定key值，通过轮询的方式来决定数据去哪一个分区

//没有给定分区，也没有给数据的key值，那么就会使用轮循的方式实现分区  
producer.send(**new** ProducerRecord<String, String>(**"test"**, **"hello world"**+i)); //ProducerRecord 使用两个形参，第一个形参是我们的topic主题，第二个参数就是我们需要发送的消息

第四种：自定义分区

public class MyOwnPartitioner implements Partitioner{  
  
 /\*\*  
 \* 这个方法决定了我们的数据的分区的方式，  
 \* @param topic  
 \* @param key  
 \* @param keyBytes  
 \* @param value  
 \* @param valueBytes  
 \* @param cluster  
 \* @return  
 \*/  
 public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {  
  
 //返回值决定了数据去哪一个分区  
 return 2;  
 }  
  
 public void close() {  
  
 }  
  
 public void configure(Map<String, ?> configs) {  
  
 }  
}

props.put("partitioner.class","cn.itcast.kafkaStudy.MyOwnPartitioner");

#### 4.4.2 数据存储机制

partition：分区的概念，一个topic当中的消息，可以拆分成多个partition分区，存放在多个不同的服务器上，实现数据存放的横向扩展

repliaction：副本，所有的partition都可以指定存放几个副本，做到数据的冗余，保证数据的安全

#### 4.4.3 消息不丢失机制

生产者producer：分为同步模式与异步模式，同步模式效率低，异步模式效率高

具体配置如下

kafka的ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到

如果是同步模式：ack机制能够保证数据的不丢失，如果ack设置为0，风险很大，一般不建议设置为0

producer.type=sync   
request.required.acks=1

如果是异步模式：通过buffer来进行控制数据的发送，有两个值来进行控制，时间阈值与消息的数量阈值，如果buffer满了数据还没有发送出去，如果设置的是立即清理模式，风险很大，一定要设置为阻塞模式

结论：producer有丢数据的可能，但是可以通过配置保证消息的不丢失

producer.type=async   
request.required.acks=1   
queue.buffering.max.ms=5000   
queue.buffering.max.messages=10000   
queue.enqueue.timeout.ms = -1   
batch.num.messages=200

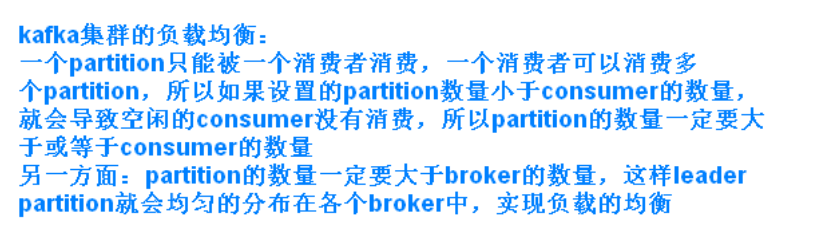
消费者consumer：通过offset来记录每次消费到了哪一条数据，

低速的消费模式：offset记录在了本地磁盘文件

高速的消费模式：zookeeper的节点上

服务器broker：数据分区，备份保证数据的不丢失

#### 消费者的负载均衡机制



#### kafka的命令行的使用

创建topic

./kafka-topics.sh --create --partitions 3 --replication-factor 2 --topic test --zookeeper node01:2181,node02:2181,node03:2181

查看所有的topic

./kafka-topics.sh --list --zookeeper node01:2181,node02:2181,node03:2181

kafka的消息发送

./kafka-console-producer.sh --broker-list node01:9092,node02:9092,node03:9092 --topic test

kafka消息的消费

./kafka-console-consumer.sh --bootstrap-server node01:9092,node02:9092,node03:9092 --from-beginning --topic test

使用zk来连接集群

./kafka-console-consumer.sh --zookeeper node01:2181,node02:2181,node03:2181 --from-beginning --topic test

## kafka的API使用

第一步：创建maven工程，导入jar包

<**dependencies**>  
 *<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->* <**dependency**>  
 <**groupId**>org.apache.kafka</**groupId**>  
 <**artifactId**>kafka-clients</**artifactId**>  
 <**version**>1.0.0</**version**>  
 </**dependency**>  
</**dependencies**>

第二步：kafka的生产者API

**public class** KafkaProducerStudy {  
 *//通过javaAPI操作kafka的生产者，往test这个topic里面生产消息* **public static void** main(String[] args) {  
 Properties props = **new** Properties();  
 props.put(**"bootstrap.servers"**, **"node01:9092,node02:9092,node03:9092"**);  
 props.put(**"acks"**, **"all"**); *//kafka的一个消息确认机制，确保消息的不丢失* props.put(**"retries"**, 0);  
 props.put(**"batch.size"**, 16384);  
 props.put(**"linger.ms"**, 1);  
 props.put(**"buffer.memory"**, 33554432);  
 props.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
 props.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
  
 Producer<String, String> producer = **new** KafkaProducer<String, String>(props);  
 **for** (**int** i = 0; i < 100; i++){  
 producer.send(**new** ProducerRecord<String, String>(**"test"**, **"hello world"**+i)); *//ProducerRecord 使用两个形参，第一个形参是我们的topic主题，第二个参数就是我们需要发送的消息* }  
 producer.close();  
 }  
  
  
  
  
}

第三步：kafka的消费者的API

自动管理offset

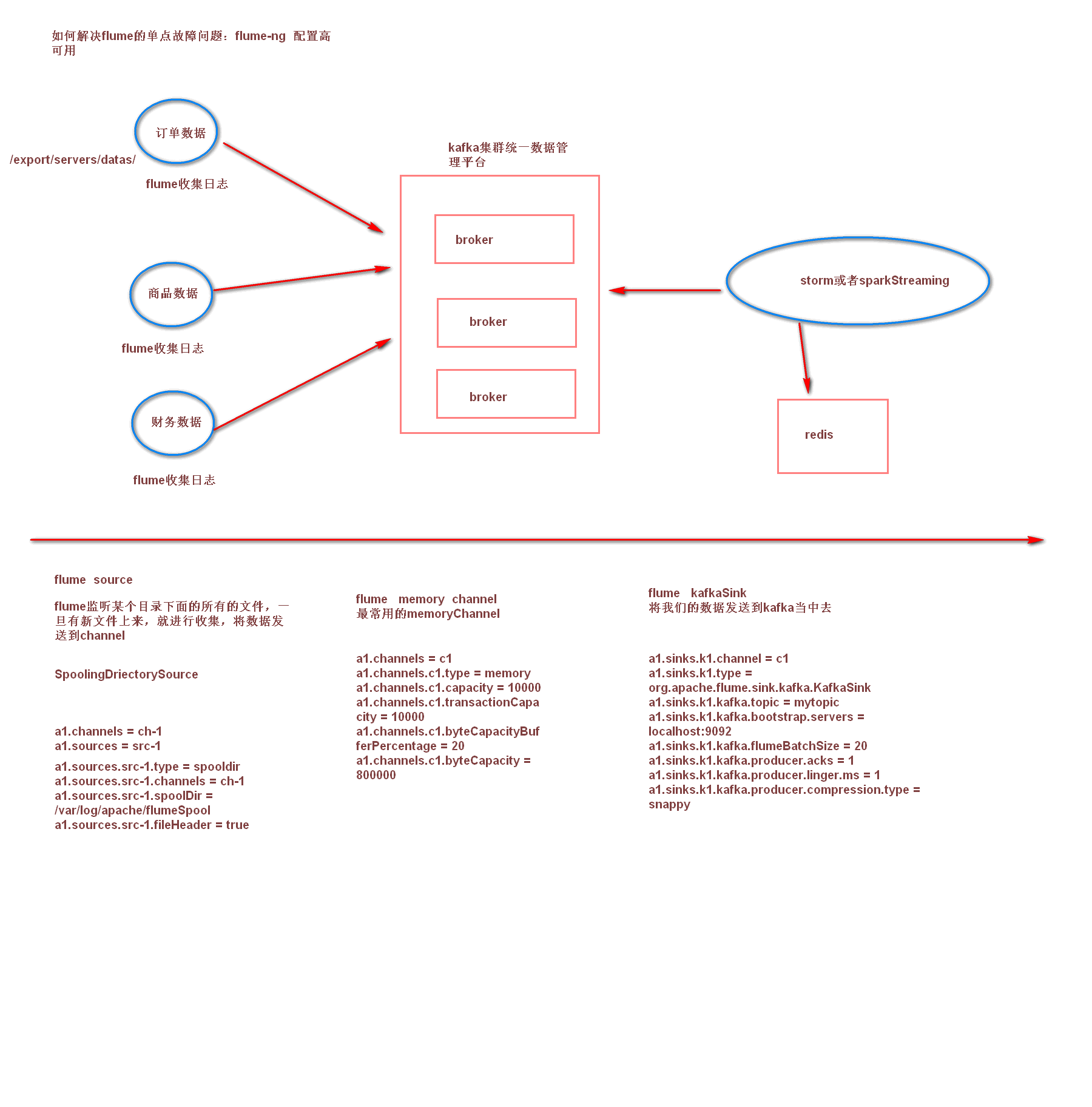
**public class** KafkaProducerStudy {  
 *//通过javaAPI操作kafka的生产者，往test这个topic里面生产消息* **public static void** main(String[] args) {  
 Properties props = **new** Properties();  
 props.put(**"bootstrap.servers"**, **"node01:9092,node02:9092,node03:9092"**);  
 props.put(**"acks"**, **"all"**); *//kafka的一个消息确认机制，确保消息的不丢失* props.put(**"retries"**, 0);  
 props.put(**"batch.size"**, 16384);  
 props.put(**"linger.ms"**, 1);  
 props.put(**"buffer.memory"**, 33554432);  
 props.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
 props.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
  
 Producer<String, String> producer = **new** KafkaProducer<String, String>(props);  
 **for** (**int** i = 0; i < 100; i++){  
 producer.send(**new** ProducerRecord<String, String>(**"test"**, **"hello world"**+i)); *//ProducerRecord 使用两个形参，第一个形参是我们的topic主题，第二个参数就是我们需要发送的消息* }  
 producer.close();  
 }  
}

kafka的手动管理offset

**public static void** main(String[] args) {  
 Properties props = **new** Properties();  
 props.put(**"bootstrap.servers"**, **"node01:9092,node02:9092,node03:9092"**);  
 props.put(**"group.id"**, **"test"**);  
 props.put(**"enable.auto.commit"**, **"false"**);*//如果需要手动管理offset，一定要注意，这个配置要给false* props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  
 KafkaConsumer<String, String> consumer = **new** KafkaConsumer<String,String>(props);  
 consumer.subscribe(Arrays.*asList*(**"test"**));  
 **final int** minBatchSize = 10;  
 List<ConsumerRecord<String, String>> buffer = **new** ArrayList <ConsumerRecord<String, String>>();  
 **while** (**true**) {  
 ConsumerRecords<String, String> records = consumer.poll(100);*//拉取数据* **for** (ConsumerRecord<String, String> record : records) {  
 buffer.add(record);  
 }  
 **if** (buffer.size() >= minBatchSize) {  
 *// insertIntoDb(buffer);实现自己的业务逻辑在这里* consumer.commitSync();*//一批次的提交我们的offset* buffer.clear();  
 }  
 }  
}

## kafka与其他的整合使用

#### 4.6.1 flume与kafka的整合



实现flume监控某个目录下面的所有文件，然后将文件收集发送到kafka消息系统中

第一步：flume下载地址

http://archive.apache.org/dist/flume/1.8.0/apache-flume-1.8.0-bin.tar.gz

第二步：上传解压flume

第三步：配置flume.conf

#为我们的source channel sink起名

a1.sources = r1

a1.channels = c1

a1.sinks = k1

#指定我们的source收集到的数据发送到哪个管道

a1.sources.r1.channels = c1

#指定我们的source数据收集策略

a1.sources.r1.type = spooldir

a1.sources.r1.spoolDir = /export/servers/flumedata

a1.sources.r1.deletePolicy = never

a1.sources.r1.fileSuffix = .COMPLETED

a1.sources.r1.ignorePattern = ^(.)\*\\.tmp$

a1.sources.r1.inputCharset = GBK

#指定我们的channel为memory,即表示所有的数据都装进memory当中

a1.channels.c1.type = memory

#指定我们的sink为kafka sink，并指定我们的sink从哪个channel当中读取数据

a1.sinks.k1.channel = c1

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.kafka.topic = test

a1.sinks.k1.kafka.bootstrap.servers = node01:9092,node02:9092,node03:9092

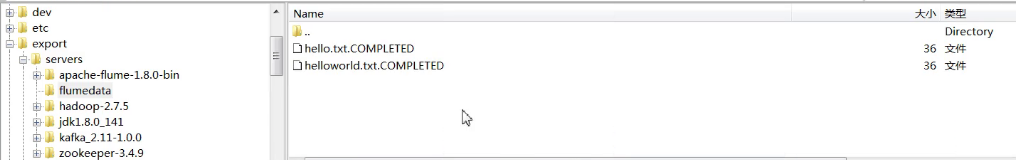
a1.sinks.k1.kafka.flumeBatchSize = 20

a1.sinks.k1.kafka.producer.acks = 1

启动flume

bin/flume-ng agent --conf conf --conf-file conf/flume.conf --name a1 -Dflume.root.logger=INFO,console

把文件放到flumeData文件夹下,刷新,文件会自动变为.COMPLETED文件



然后在kafka中开启消费者,就会看到文件中的数据.

#### 4.6.2 storm与kafka集成

##### 旧版本的kafka与storm之间相互集成

第一步：导入jar包

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>1.1.1</version>

</dependency>

<!-- use old kafka spout code -->

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-kafka</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka\_2.10</artifactId>

<version>0.8.2.1</version>

<exclusions>

<exclusion>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</exclusion>

<exclusion>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

</exclusion>

</exclusions>

</dependency>

第二步：代码实现

topologyBuilder.setSpout("WordCountFileSpout", new KafkaSpout(new SpoutConfig(new ZkHosts("zk01:2181,zk02:2181,zk03:2181"),"test","/test","storm")), 1);

##### 新版本的kafka与storm1.1.1集成

第一步：导入jar包

<!-- use new kafka spout code -->

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-kafka-client</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>0.10.0.0</version>

</dependency>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>1.1.1</version>

<scope>provided</scope>

</dependency>

第二步：编写我们的主函数入口程序

**public** **class** KafkStormTopo {

**public** **static** **void** main(String[] args) **throws** Exception {

KafkaSpoutConfig.Builder<String, String> builder = KafkaSpoutConfig.*builder*("192.168.52.200:9092,192.168.52.201:9092,192.168.52.202:9092","yun01");

builder.setGroupId("test\_storm\_wc");

KafkaSpoutConfig<String, String> kafkaSpoutConfig = builder.build();

TopologyBuilder topologyBuilder = **new** TopologyBuilder();

topologyBuilder.setSpout("WordCountFileSpout",**new** KafkaSpout<String,String>(kafkaSpoutConfig), 1);

topologyBuilder.setBolt("readKafkaBolt", **new** KafkaBolt()).shuffleGrouping("WordCountFileSpout");

Config config = **new** Config();

**if**(args !=**null** && args.length > 0){

config.setDebug(**false**);

StormSubmitter submitter = **new** StormSubmitter();

submitter.*submitTopology*("kafkaStromTopo", config, topologyBuilder.createTopology());

}**else**{

config.setDebug(**true**);

LocalCluster cluster = **new** LocalCluster();

cluster.submitTopology("kafkaStromTopo", config, topologyBuilder.createTopology());

}

}

}

第三步：开发我们的kafkabolt作为消息处理

**public** **class** KafkaBolt **extends** BaseBasicBolt {

@Override

**public** **void** execute(Tuple input, BasicOutputCollector collector) {

System.***out***.println(input.getValues().get(4)+"消息接受bolt");

}

@Override

**public** **void** declareOutputFields(OutputFieldsDeclarer declarer) {

}

}

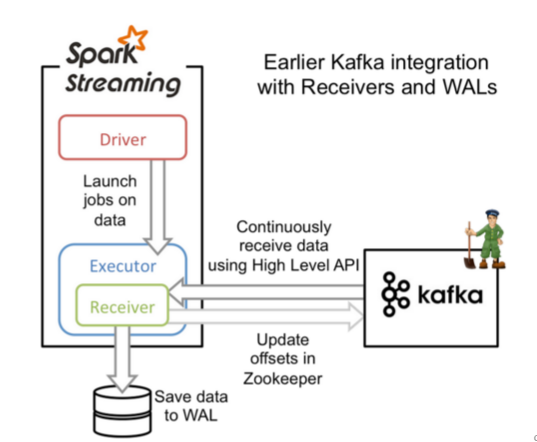
#### 4.6.3 Spark Streaming整合kafka实战

kafka作为一个实时的分布式消息队列，实时的生产和消费消息，这里我们可以利用SparkStreaming实时地读取kafka中的数据，然后进行相关计算。

在Spark1.3版本后，KafkaUtils里面提供了两个创建dstream的方法，一种为KafkaUtils.createDstream，另一种为KafkaUtils.createDirectStream。

##### KafkaUtils.createDstream方式

KafkaUtils.createDstream(ssc, [zk], [group id], [per-topic,partitions] ) 使用了receivers接收器来接收数据，利用的是Kafka高层次的消费者api，对于所有的receivers接收到的数据将会保存在[Spark](http://lib.csdn.net/base/spark" \t "http://blog.csdn.net/zhongguozhichuang/article/details/_blank" \o "Apache Spark知识库) executors中，然后通过Spark Streaming启动job来处理这些数据，默认会丢失，可启用WAL日志，它同步将接受到数据保存到分布式文件系统上比如HDFS。 所以数据在出错的情况下可以恢复出来 。



A、创建一个receiver接收器来对kafka进行定时拉取数据，这里产生的dstream中rdd分区和kafka的topic分区不是一个概念，故如果增加特定主体分区数仅仅是增加一个receiver中消费topic的线程数，并没有增加spark的并行处理的数据量。  
B、对于不同的group和topic可以使用多个receivers创建不同的DStream   
C、如果启用了WAL(spark.streaming.receiver.writeAheadLog.enable=true)

同时需要设置存储级别(默认StorageLevel.MEMORY\_AND\_DISK\_SER\_2)，

##### KafkaUtils.createDstream实战

1. 添加kafka的pom依赖

|  |
| --- |
| <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-streaming-kafka\_0-8\_2.11</artifactId>  <version>2.0.2</version> </dependency> |

1. 启动zookeeper集群

**zkServer.sh start**

1. 启动kafka集群

**kafka-server-start.sh /export/servers/kafka/config/server.properties**

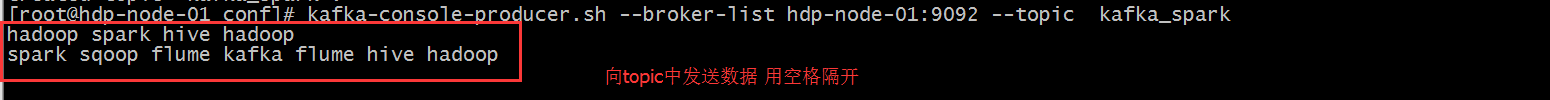
1. 创建topic

**kafka-topics.sh --create --zookeeper hdp-node-01:2181 --replication-factor 1 --partitions 3 --topic kafka\_spark**

1. 向topic中生产数据

通过shell命令向topic发送消息

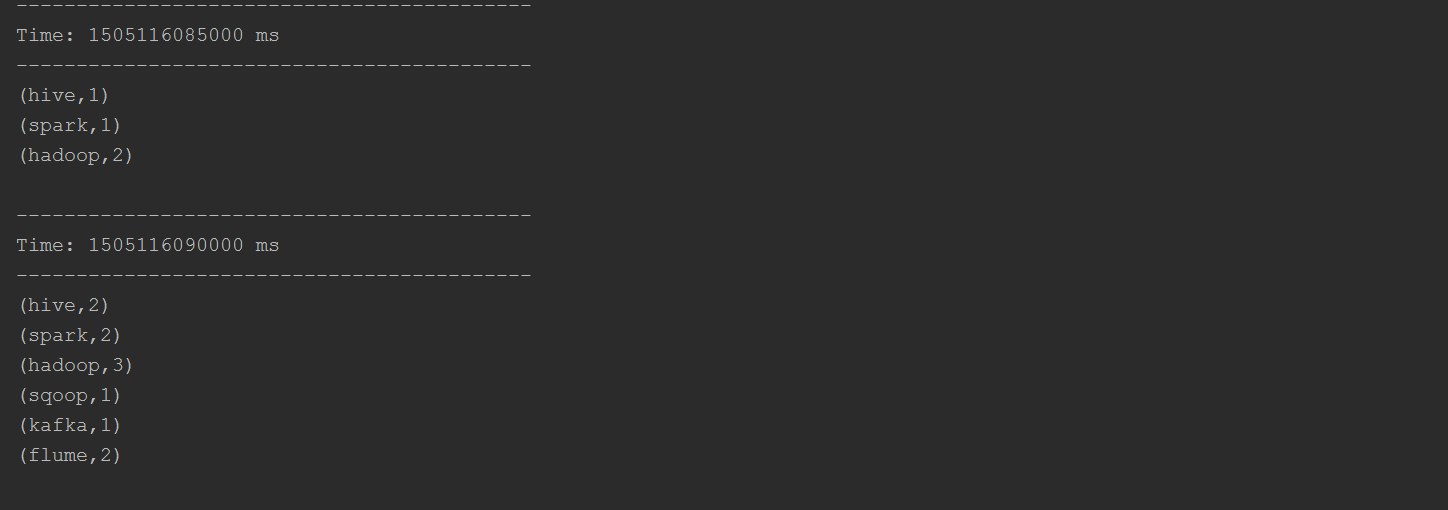
**kafka-console-producer.sh --broker-list hdp-node-01:9092 --topic kafka\_spark**



1. 编写Spark Streaming应用程序

|  |
| --- |
| **package** cn.itcast.dstream.kafka **import** org.apache.spark.{SparkConf, SparkContext} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.streaming.dstream.{DStream, ReceiverInputDStream} **import** org.apache.spark.streaming.kafka.KafkaUtils **import** scala.collection.immutable  //*todo:利用sparkStreaming对接kafka实现单词计数----采用receiver(高级API)* **object** SparkStreamingKafka\_Receiver {  **def** main(args: Array[String]): Unit = {  //1、创建sparkConf  **val** sparkConf: SparkConf = **new** SparkConf()  .setAppName("SparkStreamingKafka\_Receiver")  .setMaster("local[4]")  .set("spark.streaming.receiver.writeAheadLog.enable","true") //开启wal预写日志，保存数据源的可靠性  //2、创建sparkContext  **val** sc = **new** SparkContext(sparkConf)  sc.setLogLevel("WARN")  //3、创建StreamingContext  **val** ssc = **new** StreamingContext(sc,*Seconds*(5))  //设置checkpoint  ssc.checkpoint("./Kafka\_Receiver")  //4、定义zk地址  **val** zkQuorum="node1:2181,node2:2181,node3:2181"  //5、定义消费者组  **val** groupId="spark\_receiver1"  //6、定义topic相关信息 Map[String, Int]  // 这里的value并不是topic分区数，它表示的topic中每一个分区被N个线程消费  **val** topics=*Map*("spark\_kafka" -> 2)  //7、通过KafkaUtils.createStream对接kafka  //这个时候相当于同时开启3个receiver接受数据  **val** receiverDstream: immutable.IndexedSeq[ReceiverInputDStream[(String, String)]] = (1 to 3).map(x => {  **val** stream: ReceiverInputDStream[(String, String)] = KafkaUtils.*createStream*(ssc, zkQuorum, groupId, topics)  stream  }  )  //使用ssc.union方法合并所有的receiver中的数据  **val** unionDStream: DStream[(String, String)] = ssc.union(receiverDstream)   //8、获取topic中的数据  **val** topicData: DStream[String] = unionDStream.map(\_.\_2)  //9、切分每一行,每个单词计为1  **val** wordAndOne: DStream[(String, Int)] = topicData.flatMap(\_.split(" ")).map((\_,1))  //10、相同单词出现的次数累加  **val** result: DStream[(String, Int)] = wordAndOne.reduceByKey(\_+\_)  //11、打印输出  result.print()  //开启计算  ssc.start()  ssc.awaitTermination()  }  } |

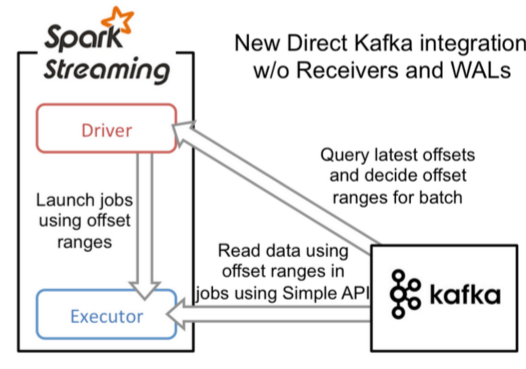
1. 运行代码,查看控制台结果数据



总结:

通过这种方式实现，刚开始的时候系统正常运行，没有发现问题，但是如果系统异常重新启动sparkstreaming程序后，发现程序会重复处理已经处理过的数据，这种基于receiver的方式，是使用Kafka的高级API，topic的offset偏移量在ZooKeeper中。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据只被处理一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。官方现在也已经不推荐这种整合方式，我们使用官网推荐的第二种方式kafkaUtils的createDirectStream()方式。

##### KafkaUtils.createDirectStream方式

这种方式不同于Receiver接收数据，它定期地从kafka的topic下对应的partition中查询最新的偏移量，再根据偏移量范围在每个batch里面处理数据，Spark通过调用kafka简单的消费者Api（低级api）读取一定范围的数据。  
相比基于Receiver方式有几个优点：   
A、简化并行

不需要创建多个kafka输入流，然后union它们，sparkStreaming将会创建和kafka分区数相同的rdd的分区数，而且会从kafka中并行读取数据，spark中RDD的分区数和kafka中的topic分区数是一一对应的关系。

B、高效，

第一种实现数据的零丢失是将数据预先保存在WAL中，会复制一遍数据，会导致数据被拷贝两次，第一次是接受kafka中topic的数据，另一次是写到WAL中。而没有receiver的这种方式消除了这个问题。   
C、恰好一次语义(Exactly-once-semantics)

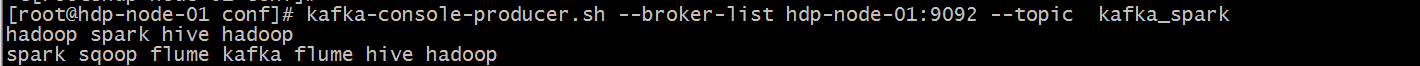
Receiver读取kafka数据是通过kafka高层次api把偏移量写入zookeeper中，虽然这种方法可以通过数据保存在WAL中保证数据不丢失，但是可能会因为sparkStreaming和ZK中保存的偏移量不一致而导致数据被消费了多次。EOS通过实现kafka低层次api，偏移量仅仅被ssc保存在checkpoint中，消除了zk和ssc偏移量不一致的问题。缺点是无法使用基于zookeeper的kafka监控工具。

##### KafkaUtils.createDirectStream实战

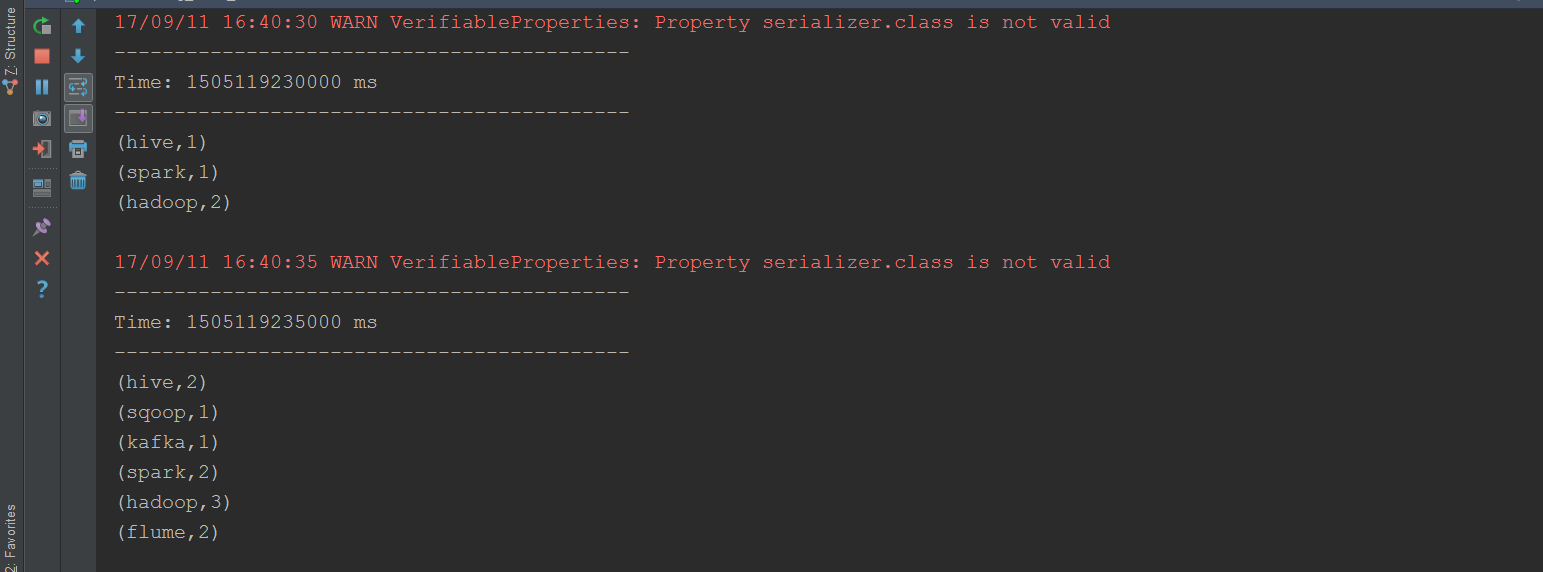
|  |
| --- |
| **package** cn.itcast.dstream.kafka  **import** kafka.serializer.StringDecoder **import** org.apache.spark.{SparkConf, SparkContext} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.streaming.dstream.{DStream, InputDStream} **import** org.apache.spark.streaming.kafka.KafkaUtils  //*todo:利用sparkStreaming对接kafka实现单词计数----采用Direct(低级API)* **object** SparkStreamingKafka\_Direct {  **def** main(args: Array[String]): Unit = {  //1、创建sparkConf  **val** sparkConf: SparkConf = **new** SparkConf()  .setAppName("SparkStreamingKafka\_Direct")  .setMaster("local[2]")  //2、创建sparkContext  **val** sc = **new** SparkContext(sparkConf)  sc.setLogLevel("WARN")  //3、创建StreamingContext  **val** ssc = **new** StreamingContext(sc,*Seconds*(5))  ssc.checkpoint("./Kafka\_Direct")  //4、配置kafka相关参数  **val** kafkaParams=*Map*("metadata.broker.list"->"node1:9092,node2:9092,node3:9092","group.id"->"Kafka\_Direct")  //5、定义topic  **val** topics=*Set*("spark01")  //6、通过 KafkaUtils.createDirectStream接受kafka数据，这里采用是kafka低级api偏移量不受zk管理  **val** dstream: InputDStream[(String, String)] = KafkaUtils.*createDirectStream*[String,String,StringDecoder,StringDecoder](ssc,kafkaParams,topics)  //7、获取kafka中topic中的数据  **val** topicData: DStream[String] = dstream.map(\_.\_2)  //8、切分每一行,每个单词计为1  **val** wordAndOne: DStream[(String, Int)] = topicData.flatMap(\_.split(" ")).map((\_,1))  //9、相同单词出现的次数累加  **val** result: DStream[(String, Int)] = wordAndOne.reduceByKey(\_+\_)  //10、打印输出  result.print()  //开启计算  ssc.start()  ssc.awaitTermination()  } } |

1. 查看对应的效果

向topic中添加数据



查看控制台的输出：



#### 4.6.4 kafka与Hbase的集成使用

##### Kafka接收数据,并消费到Hbase数据库

一、

   1、生产者 产生数据

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package kafakaTohbase;    import java.util.Properties;    import kafka.javaapi.producer.Producer;  import kafka.producer.KeyedMessage;  import kafka.producer.ProducerConfig;    public class KafkaProducer {        public static void main(String[] args) {          Properties props = new Properties();          props.put("zk.connect", KafkaProperties.zkConnect);          props.put("serializer.class", "kafka.serializer.StringEncoder");          props.put("metadata.broker.list", "hdjt01:9092,hdjt02:9092,hdjt03:9092");          ProducerConfig config = new ProducerConfig(props);          Producer<String, String> producer = new Producer<String, String>(config);          for (int i = 0; i < 10; i++){            producer.send(new KeyedMessage<String, String>("test5", "liu" + i));      }      }      } |

　　注： props.put("serializer.class", "kafka.serializer.StringEncoder")  发送的数据是String,

          还可以是 二进制数组形式：

props.put("serializer.class", "kafka.serializer.DefaultEncoder");  
props.put("key.serializer.class", "kafka.serializer.StringEncoder");  如果没有这个，就代表 key也是二进制形式。

         生产者发送的都是keyvalue对

     2、消费者

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66 | package kafakaTohbase;    import java.io.IOException;  import java.util.HashMap;  import java.util.List;  import java.util.Map;  import java.util.Properties;    import kafka.consumer.ConsumerConfig;  import kafka.consumer.ConsumerIterator;  import kafka.consumer.KafkaStream;  import kafka.javaapi.consumer.ConsumerConnector;  import kafka.javaapi.producer.Producer;  import kafka.producer.KeyedMessage;  import kafka.producer.ProducerConfig;    public class KafkaConsumer extends Thread{         private final ConsumerConnector consumer;       private final String topic;            public KafkaConsumer(String topic) {              consumer = kafka.consumer.Consumer                      .createJavaConsumerConnector(createConsumerConfig());              this.topic = topic;          }            private static ConsumerConfig createConsumerConfig() {              Properties props = new Properties();              props.put("zookeeper.connect", KafkaProperties.zkConnect);              props.put("group.id", KafkaProperties.groupId1);              props.put("zookeeper.session.timeout.ms", "40000");       //zookeeper 与 region server 的链接超时时间              props.put("zookeeper.sync.time.ms", "200");              props.put("auto.commit.interval.ms", "1000"); <br>               //props.put("auto.offset.reset", "smallest");//可以读取旧数据，默认不读取              return new ConsumerConfig(props);          }            @Override          public void run() {                Map<String, Integer> topicCountMap = new HashMap<String, Integer>();              topicCountMap.put(topic, new Integer(1));              Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap = consumer                      .createMessageStreams(topicCountMap);              KafkaStream<byte[], byte[]> stream = consumerMap.get(topic).get(0);              ConsumerIterator<byte[], byte[]> it = stream.iterator();              HBaseUtils hbase = new HBaseUtils();              while (it.hasNext()) {  //相当于加了一把锁，一直返回true  //              System.out.println("3receive：" + it.next().message());                  try {                      System.out.println("11111");                      hbase.put(new String(it.next().message()));                  } catch (IOException e) {                      // TODO Auto-generated catch block                      e.printStackTrace();                  }    //            try {  //                sleep(300);    // 每条消息延迟300ms  //            } catch (InterruptedException e) {  //                e.printStackTrace();  //            }              }          }    } |

　　连接hbase，配置信息

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package kafakaTohbase;    import java.io.IOException;  import java.util.Random;    import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.hbase.HBaseConfiguration;  import org.apache.hadoop.hbase.client.HTable;  import org.apache.hadoop.hbase.client.Put;  import org.apache.hadoop.hbase.util.Bytes;    public class HBaseUtils {       public  void put(String string) throws IOException {              //设置HBase据库的连接配置参数              Configuration conf = HBaseConfiguration.create();              conf.set("hbase.zookeeper.quorum",  "hdjt01:2181,hdjt02:2181,hdjt03:2181");  //  Zookeeper的地址  //          conf.set("hbase.zookeeper.property.clientPort", "42182");              Random random = new Random();              long a = random.nextInt(1000000000);              String tableName = "emp";              String rowkey = "rowkey"+a ;              String columnFamily = "basicinfo";              String column = "empname";              //String value = string;              HTable table=new HTable(conf, tableName);              Put put=new Put(Bytes.toBytes(rowkey));              put.add(Bytes.toBytes(columnFamily), Bytes.toBytes(column), Bytes.toBytes(string));              table.put(put);//放入表              System.out.println("放入成功");              table.close();//释放资源              }  } |

　　测试消费者：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class Kafkaceshi {        public static void main(String[] args) {  //      KafkaProducer a=new KafkaProducer ();  //      a.producer();          KafkaConsumer consumerThread = new KafkaConsumer(KafkaProperties.topic);          consumerThread.run();        }    } |